Chapter III:
Visualization Techniques Framework

A Concrete Instantiable Description

In the previous chapter we discussed visualization techniques at an abstract, functional level. This level of specification is useful because it captures the general purpose and capabilities of a technique. The functional specification also allows comparisons of techniques without the added complexity of interface and structural details. This level of specification however is insufficient to capture a technique in enough detail for instantiation (i.e. rendering it as a visualization interface). In order to increase the applicability and practicality of our framework, we must also describe how the functional specifications described in the previous chapter can be augmented to an instantiable form.

Other previous frameworks on visualization techniques either focussed on the functional level (e.g. [*Tweedie, 1997*], [*Card et al., 1997*], [*Card et al., 1999*]) or the instantiation level (e.g. Data explorer, IRIS Explorer, AVS [*Brodlie, 1991*]) in isolation. In this thesis it was crucial for us to describe a framework that encapsulates both levels of description. The functional level informs our automatic design system of the primary primitives within a visualization technique, their uses, and how these primitives may be combined with each other. The instantiation level allows our designer to describe the structural details of a technique (e.g. data and graphical elements, input-devices, etc) so that the technique can be rendered as an active interface. By including both levels in the same framework as well as a method of transition from one level to the other we provide the advantages of both types of description without requiring a change in the conceptual model or descriptive language as we would with previous frameworks. In addition this two level design methodology is also useful to visualization designers because it encourages them to initially focus on the functional aspects of visualization techniques, free from structural constraints. Once the functionality has been fully designed, a designer may enrich the technique with structural detail. As a result designers will be less apt to falsely constrain functional capabilities as a result of structural concerns. For a more complete discussion of how our framework differs from previous work refer to appendix B-1.

In this chapter we present a five step process, which we call the *instantiation augmentation process*, that can convert any functional specification into an instantiable visualization technique (Figure III-1[1]). A *functional* specification (like the ones shown in the previous chapter) captures the "core" functionality[2] of a technique in as general and abstract terms as possible. An *instantiation* specification augments a *functional* specification with specific input-devices and translation functions[3] that are required to make the technique operational. The five steps in the instantiation augmentation process dictate the general look or structure of the technique but does not change its underlying functionality. For example, the topmost diagram in Figure III-1 shows a functional specification for the object highlight technique, which graphically transforms a set of user selected objects. The functional specification is very simple consisting of an enumeration selection method followed by a graphical transform. The instantiation specification of this technique (shown at the bottom of Figure III-1) is more complex, consisting of various input-devices (e.g. *bounding-box*) and translation functions (e.g. *get-values* function) in addition to the object definition and transformation functions used in the functional space.

Next, we describe the five steps of the *instantiation augmentation process* shown in Figure III-1. The gray highlights in Figure III-1 indicate changes made to the specification at each step.

Step1. *Pick specific object selection and transformation functions*:

We must decide which specific object definition and transformation functions to use from the broad general classes (e.g. *enumeration, functional description, data transform, mapping transform, graphical transform, rendering transform*) described in the previous chapter. For example in Figure III-1 we use the *assign* function which is a specific instance of the more general graphical transform class. There are no sub-class functions to the enumeration object definition class so no further refinements are needed in that component. We consider this first step as part of the instantiation augmentation process but it can sometimes be included in the functional specification process if specific transform instances (e.g. *assign*) are required to define the "core" functionality of a technique. I.e. a functional specification may consist of broad function classes (e.g. *enumeration, graphical transform*) or of specific function instances (e.g. *assign*). However, a functional specification should be stated in as general and abstract terms as possible. Thus, function instances should only be used when absolutely necessary.

---

[1] The specifications shown in Figure III-1 and in this chapter uses the same notation as the functional specifications (i.e. *ODT* or *object-definition-transformation* diagrams) shown in the previous chapter. Details on the diagrammatic notations used in these specifications can be found in appendix A-1.

[2] In this work we assert that "core" functionality is captured by either object definition or transformation functions and nothing else. Translation functions and input-devices are considered part of the structural (as opposed to functional) aspect of a visualization technique.

[3] Translation functions transform objects between the data and graphical realms and also among the different object types within each realm. More detail on these functions is given in the following sections.

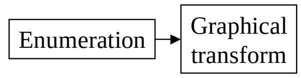
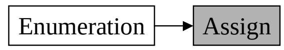
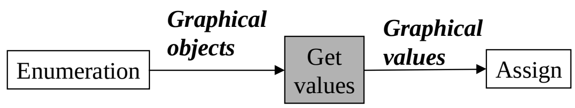
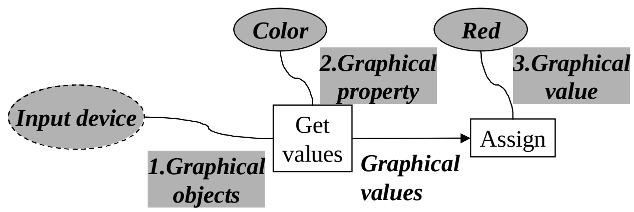
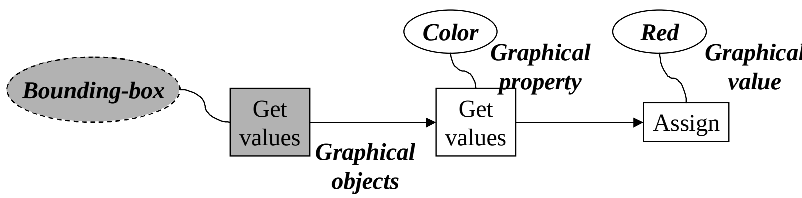
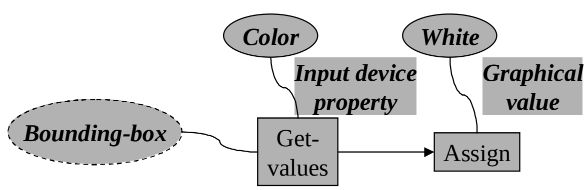
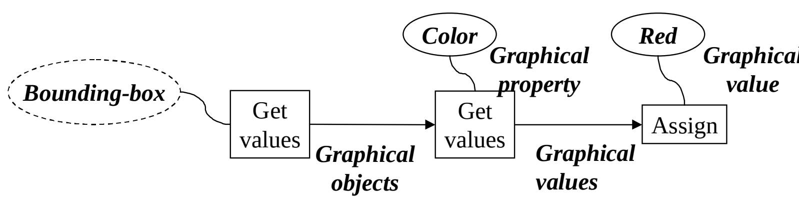
Functional Specification	
Step 1:	*Pick specific object selection and transformation functions* 
Step 2:	*Match outputs with inputs through translation functions* 
Step 3:	*Determine how function inputs are chosen* 
Step 4:	*Determine input-devices* 
Step 5: Instantiation Specification	*Specify initialization arguments to the input-devices* <u>*Input-device initialization specification*</u>: Color property value of *bounding-box* input-device is set to *white*.  <u>*Main technique specification*</u>: This is the complete specification from Step 4. 

Figure III-1: Diagrammatic representation of the five-step instantiation augmentation process for the object highlight technique. Additions made in each step are shown in gray.

Step2. *Match outputs with inputs through translation functions*:

A visualization consists of elements from three different realms, the data realm, the graphical realm and the output media. The transformation and object definition functions within a visualization technique may operate on objects, attributes, and values in any of these realms. Proper operation of a technique depends on whether we apply its object definition and transformation functions to the correct realm and to the correct element types within each realm. A set of *translation functions* is available to translate the outputs of functions and devices between the realms. These translation functions are also used to explore object relationships and query for object state so that we may obtain the correct element types within each realm. At this point in the instantiation augmentation process, we decide which translation functions need to be inserted between the object definition and transformation functions declared in step 1 to ensure that the outputs of the object definition function match the inputs of the following transformation function. For example in Figure III-1, step 2, the *enumeration* object definition step produces a set of graphical objects. We use the *get-values* translation function to extract a set of *color* graphical values from these objects because graphical values are needed in the subsequent *assign* graphical transform.

Step3. *Determine how function inputs are chosen*:

Object definition, transformation, and translation functions all have inputs and outputs. The input arguments to these functions can be provided by other functions, such as a translation function (step 2), an object definition function or a transformation function (as is outlined in chapter II-2). Input arguments that are not provided by a function must be preset by a visualization designer or controlled by a user through an input-device. Here we decide between these two alternatives. For example in Figure III-1, step 3, the object highlighting technique has three unspecified input arguments after the first two steps. These arguments include: 1) the set of graphical objects to feed into the *get-values* function, 2) a graphical property for the *get-values* function, and 3) a property value for the *assign* transform. In this example, we provide the latter two values through designer defaults. The graphical property is set to *color* and the property value is set to *red*. The first input argument is obtained from the user through an input-device.

Step4. *Determine input-devices*:

In this step we determine the type of input-devices to use with the interactive system. An input-device is needed for every user input value specified in step 3. We must decide whether to provide several input values with a single device (composition of input-devices) or whether to use separate devices to provide separate inputs. The types of devices that are appropriate would depend on effectiveness measures such as the ones outlined by Card et. al. [*Card, 1990, Mackinlay, 1990*]. In Figure III-1 there is only one user input argument and we use a *bounding-box* that allows users to *enumerate* the set of objects they want to highlight. Note that a *get-values* translation function is used with all input-devices to extract relevant state information from them.

Step5. *Specify initialization arguments to the input-devices*:

Certain input-devices may require initialization values. For example, a *slider* input-device must be initialized with the maximum and minimum slider range. On the other hand a *scroll list* is initialized with a list of selectable entries. In the object highlight example, the *bounding-box* input-device **does not** require any initialization, however, we decided to initialize its *color* property anyway for aesthetic reasons. Note that the operations used for input-device initializations are visualization techniques themselves. In this case, the object definition component defines a *bounding-box* input-device object that is a designer enumerated object. Subsequently this object is passed to the *assign* graphical transform operator, which changes the *color* property of the *bounding-box* to *white*. For a more detailed example of the structural augmentation process refer to appendix B-3.

To perform the *instantiation augmentation process* outlined above, we must clearly define the specific object definition, transformation, and translation functions available; the set of visualization elements from the three realms (data, graphical, output media) that are manipulated by these functions; and the input-devices that can be used with these functions. In section III-1 we give detailed descriptions of the primitives within these three realms. Section III-1 is not meant as advancement to the field. Many of the visualization elements and input-devices presented have been captured in previous work, and the visualization function primitives recapitulates standard math theory that can be accomplished using any current programming language. However, this level of detail is necessary for our automatic design system because it requires a complete description of all primitives that are available for design. In addition, the primitives also give specific examples of the types of functions we would find in each transformation class and provides bounds on our framework, indicating the number of primitives that are required to capture the various visualization techniques described in this work. Section III-1 may be skipped if the reader is not interested in detailed descriptions of the primitives used by our automatic design system. In section III-2 we show how changing the instantiation specification of a technique can change its effectiveness at solving tasks and can sometimes lead to new and interesting design variations. Note that the design alternatives generated by exploration at the instantiation level is different from that of the functional level (chapter II-3) because here, we are keeping the semantics of a technique constant and only changing its structural content. As a result the technique still fulfills the same goals even though the method of interaction or the visual feedback may now be different. Finally we close the chapter by discussing the merits of our framework based on three criteria, completeness, coverage, and practicality (section III-3). These last two sections (sections III-2, III-3) are provided to validate and highlight the uses of our framework. They may be skipped if the reader is only interested in the automatic design aspects of this work.

III-1 Representation Language

In this section we present the visualization elements, functions, and devices defined in our framework for constructing visualization techniques. These primitives were picked based on common features available in current visualization methods. It is important to note that this is not a complete list, and it can never be complete because as new techniques are developed the list must necessarily grow and change. The primitives described in our framework, however, are flexible and can be used to attain a variety of visualization effects. Also note that the primitive functions have low granularity (i.e. we use simple mathematical functions). This level of granularity gives us greater flexibility in composition and allows us to express many current visualization techniques with a relatively small set of primitive building blocks. In the next sections we describe the three primary building block classes in our framework:

1) *Visualization elements or properties* which may be from the data realm, graphical realm, or output media. These objects or object properties form the inputs and outputs of the primitive functions within a visualization technique;
2) *Visualization technique primitive functions* which may come from the object definition, transformation, or translation function classes.
3) *Input-devices* that provide users with physical (e.g. *mouse clicks*) and/or virtual (e.g. *menus*) controls so that they may interact with a visualization technique.

III-1.1 Visualization Elements or Properties

Visualization techniques operate over the data concepts, graphical objects, or output media that form a visualization. In the object definition component of a visualization technique a set of these visualization objects is chosen and then transformed. In addition to the elements selected in the object definition component, primitive visualization functions often require other object or value inputs as was illustrated in Figure III-1. These other inputs provide designers with a limited means of controlling the behavior of the functions so that they can achieve a wide range of effects with a relatively small set of functions. In this section we present the representational structures used to describe abstract data concepts and graphical objects within a visualization. Our representational structures are based on previous work [*Mackinlay, 1986a, 1986b; Roth, 1990*]. We do not characterize the output media here because the focus of this thesis is only on data and mapping functions, which operate wholly in the data and graphical realms. We leave characterization of the output media for future work.

III-1.1.1 Data Concepts

The primary element in the data realm is the data concept. A data concept is a database record, very commonly represented as rows in spreadsheet programs. For example in Figure III-2 the data concepts being represented are *houses*. Each data concept is attached to a *data type* that describes the attributes and relational structure of the data concept. Data concepts having the same attributes and relational structure will be attached to the same *data type*. Every *house* data concept or record, for example, belongs to the *house-data-type* class. The *house-data-type* class describes the five *data attributes* that are attached to each *house* data concept, namely *house-selling-price, date-sold, neighborhood, date-on-market,* and *number-of-rooms*. Each data attribute commonly corresponds to a column of values in spreadsheet programs. The first three attributes of the *house* concepts are mapped to graphics in Figure III-2.

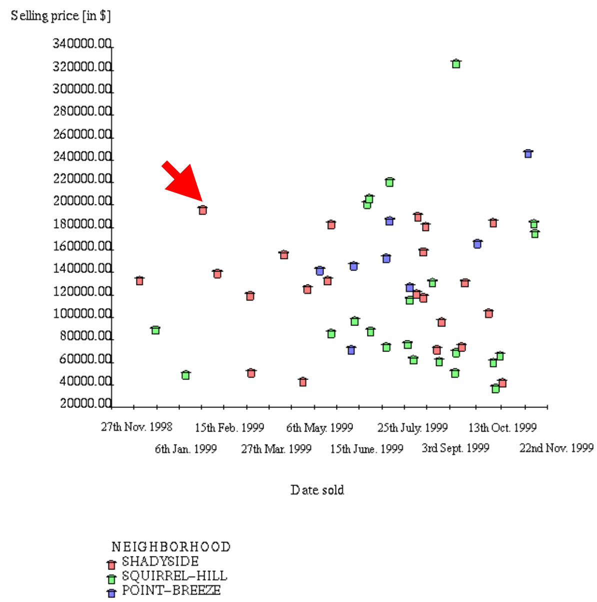


Figure III-2: Example visualization containing house data. Each mark represents a house data concept. The *x-axis* shows the *date-sold* data attribute; the *y-axis* shows the *selling-price* data attribute, and *hue* shows the *neighborhood* data attribute.

Although data concepts commonly represent "real-world" objects, e.g. a house, this need not be the case. A data concept may also represent a conceptual object, consisting of a group of related *data attributes* possibly originating from multiple different "real-world" objects. For example, we may have data concepts that contain the *house-selling-price* attribute as well as the *owner-salary* attribute. In this case the *house-selling-price* is an attribute of a house object whereas the *owner-salary* attribute is an attribute of a person. Depending on the task, our automatic system may draw data attributes from multiple different data concept classes to form new conceptual objects as required.

III-1.1.2 Graphical Objects

In a visualization, data concepts in a database are mapped to graphical objects in a graphical scene. Graphical objects, also commonly referred to as glyphs, are symbols that represent information through visual properties that are either spatial (*position-x, position-y*), retinal (*color, size*), or temporal (*jittering*). Graphical objects may be simple (e.g. *mark, bar*) consisting of only a few properties or more complex (e.g. *Chernoff Faces*[*Chernoff, 1973*], *InfoBug*[*Chuah, 1998a*]) consisting of many properties. Graphical objects commonly live within container objects (e.g. *chart*, *map*). For example, Figure III-2 shows a house data visualization. The *house* data concepts were encoded using *mark* graphical objects within a *chart* container. Containers are used to structure graphical objects (e.g. *marks*), annotation objects (e.g. *axes, axes-labels*) and other container objects. The *chart* in Figure III-2 for example is used to group the *mark* graphical objects together based on a well-defined layout scheme.

Based on previous frameworks [*Mackinlay, 1986b; Chuah*, *1995*], we organize graphical containers and objects into a hierarchy. At the top of the hierarchy is the *visualization container*. The visualization container exists within a desktop window, i.e. all the contents of the window are considered part of the visualization. For example, Figure III-3 shows a visualization of *house* data containing *charts*, *tables*, *marks, bars, axes,* and *axis-labels*. Figure III-4 shows a hierarchical breakdown of the components within the visualization in Figure III-3.

Below the visualization container are *region containers*. Regions are arranged based on very specific data attribute constraints. Two regions can be aligned (i.e. laid out side by side either horizontally or vertically) only if their common axes represent the same type of data attributes. Figure III-3, for example, has three aligned regions. Horizontal alignment is allowed here because the common axis of the three regions (*y-axis*) encodes the same data attribute (i.e. *house-address*) in all three charts.

Region containers group graphical objects together and structure them according to a layout scheme. Some example layout schemes are shown in Figure III-5. The *grid* layout, for example, constrains the positions of graphical objects so that they fall on evenly spaced rows and columns. The *chart* layout on the other hand does not have any positional constraints so that the positions of the graphical objects may be used to encode data, as in Figure III-2.

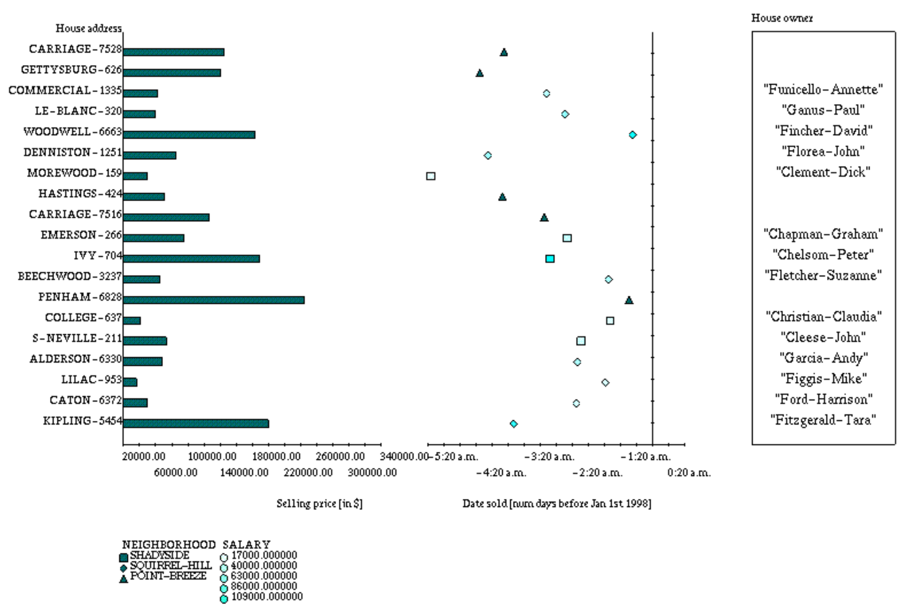


Figure III-3: Example visualization of house data. Hierarchical breakdown of graphical objects in this visualization is shown in Figure III-4. *X-axis* in left-most chart shows *selling-price*; *x-axis* in middle chart shows *date-sold, shape* shows *neighborhood,* and *saturation* shows *salary*; *text* in right-most table shows *house-owner. Y-axes* for all three regions show *house-address* data attribute.

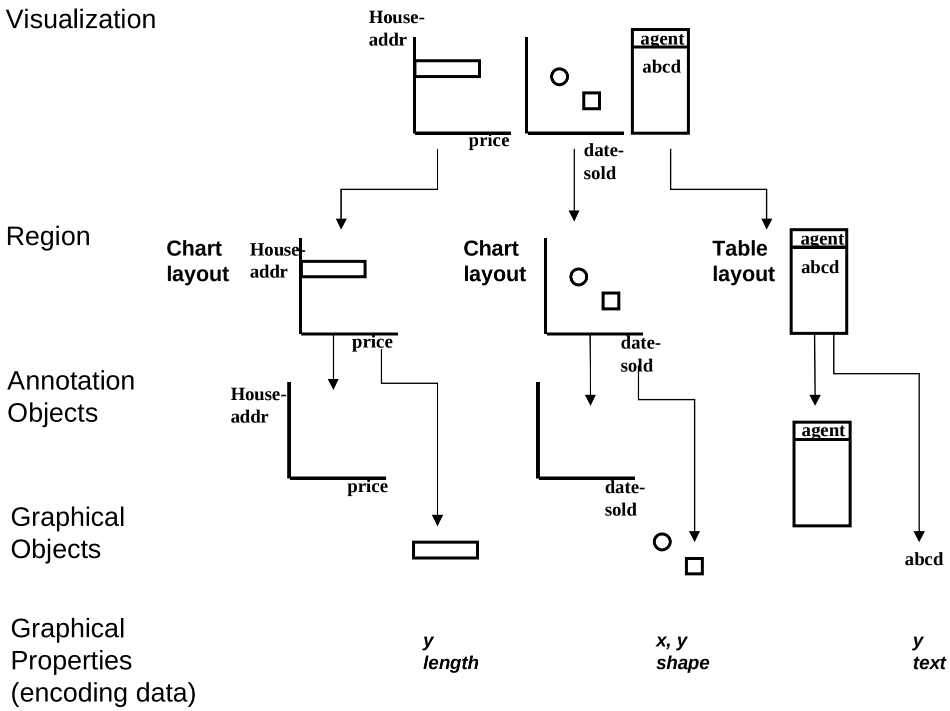


Figure III-4: Container hierarchy for visualization in Figure III-3

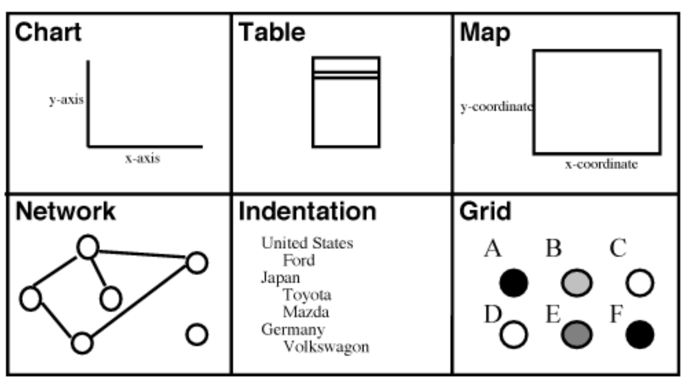


Figure III-5: Example region layout schemes (borrowed from [*Chuah, 1995*])

Annotation objects are a specialized class of graphical objects that are not containers and that **do not** represent data concepts. Some example annotation objects include chart axes, legends, and axis labels. For example, the *chart* in Figure III-2 contains a set of annotation objects including two axes, a set of axis-labels, and a legend indicating how the house neighborhoods relate to the *color* of the graphical *marks*. Annotations are used to present clarification on aspects of the visualization that are not readily apparent to users. For example, annotation objects are commonly used to show how data is encoded using graphics (e.g. *chart-axes* and *legends*), so that we can better interpret the graphical representations and tie them back to relationships within the data set. Annotations are also very useful for drawing user attention to particular graphical elements and thus are widely used for communicative purposes. For example, the *red* arrow in Figure III-2 is an annotation object for bringing user attention to a particular house object.

III-1.2 Visualization Functions

In the previous chapter we had described the general classes of visualization functions (object definition and transformation). Apart from the object definition and transformation classes, there are also translation functions that are used to convert input and output argument types. In this section we give detailed descriptions of all function primitives available to our automatic design system. Remember that all the functions described here can be accomplished with current programming languages. We enumerate the primitives here because our automatic design system requires a complete characterization of the list of functions that it can manipulate. Listing out these functions is also useful for illustrating the number and type of operators that are sufficient for describing current visualization techniques.

III-1.2.1 Object Definition Functions

In the object definition component we select a set of visualization objects for subsequent transformation. Object set definition can be performed by enumeration or through functional description. Enumeration allows the user or designer to list/enumerate all the interesting visualization objects by name. In contrast, functional description methods allow users or designers to specify an interesting set of visualization objects by applying functional constraints on the objects' attributes or properties.

Function class	**Input**	**Output**
Functional description	1. Value set, 2. Threshold value, 3. Compare operator (>, <, =, >=, <=, <>)	Boolean value set
Set operation	1. *n* object sets, 2. Set operator (*intersect, union, repeat-union, difference*)	Object set

Table III-1: List of object definition functions

Functional description functions may be simple or complex depending on the task requirements. In this thesis we use the *threshold* function as the primary functional description method. Table III-1 shows the inputs and outputs of the threshold function including all the threshold compare operators available (<, >, =, <=, >=). The threshold function has three inputs: 1) a set of input values on which to perform the threshold operation, 2) a threshold value and 3) a compare operator. It then returns a set of boolean values indicating whether each value in the input set passed the threshold. For example, if we want to find all houses whose price is above *$100k*, we would apply the threshold function to: 1) the set of *house-price* values, 2) the *$100k* threshold value, and 3) the > compare operator. The output will be a set of boolean values, indicating for each input *house-price* value whether the it exceeds *100k*. Note that the *threshold* function may be used to filter any of the elements within a visualization including: 1) data concept attributes such as *house-price* or *date-sold,* 2) graphical object properties such as *x-position* or *size,* 3) annotation object properties such as *size-of-legend, thickness of axes*, and 4) input-device properties such as *bounding-box-color* or *slider-minimum-value*.

The other functions in the object definition component are set operators. *Set operators* compose two or more objects sets together (*object-definition transformation*) to produce a single output set. Set operators are crucial for object definition composition as was illustrated in the previous chapter. Some example set operators include the *intersect, union, repeat-union* and *difference* operators. Below we apply each of these operators to three example object sets and show how their results differ.

difference ({a, b}, {b, c}, {c, d}) = *difference* ({a, c}, {c, d}) = { a, d }
intersect ({a, b}, {b, c}, {c, d}) = {}
union ({a, b}, {b, c}, {c, d}) = { a, b, c, d }
repeat-union ({a, b}, {b, c}, {c, d}) = { a, b, b, c, c, d }

The *repeat-union* operator combines all the input object sets just like the *union* operator except that it does not omit duplicate objects.

III-1.2.2 Transformation Functions

There are four classes of transformation functions, *data, mapping, graphical,* and *rendering* transforms. These four transformation classes correspond to the four main phases of the visualization creation process described in the previous chapter.

III-1.2.2.1 Mapping Transforms

Mapping transforms are the basis for visualizing data because they allow abstract data concepts to be perceived by linking them to visual graphical elements. There are two primary mapping transforms in our framework: *object mapping* and *attribute mapping* (shown in Table III-2).

Object mappings relate a class of data concepts as defined by their data type to a class of graphical objects as defined by their graphical class. Data types capture characteristics of similar data concepts that contain the same data attributes and relationships. Graphical classes capture characteristics of similar graphical objects that have a common visual appearance and contain the same graphical properties. Some common graphical classes include *bar-class, mark-class, node-class,* and *interval-bar-class*. For example, we can map all *house* data concepts to *mark* graphical objects by mapping the *house-data-type* class to the *mark-graphical-class* as in Figure III-2. As is shown in Table III-2, the first two input arguments to the *object mapping* transform are the *data-type* and *graphical class* that we want to attach.

Function class	Input	Output
Add object mapping (Delete object mapping)	1. Data type, 2. Graphical class, 3. Container object or graphical object(s)	
Add attribute mapping (Delete attribute mapping)	1. Data attribute, 2. Graphical property, 3. Container object or graphical object(s) 4. Mapping effect (*forward, backward, both*)	
Add object (Delete object)	1. Data concept(s) 2. Container object	

Table III-2: List of mapping transformation functions

The third object mapping input specifies the scope of the mapping function. The scope is defined by listing the container object (e.g. *visualization container*, *graphical space*, *region*) we want to attach the mapping to. Applying a mapping transform to a container object will cause all other containers encapsulated within it (based on the object hierarchy in Figure III-4) to inherit the mapping as well.

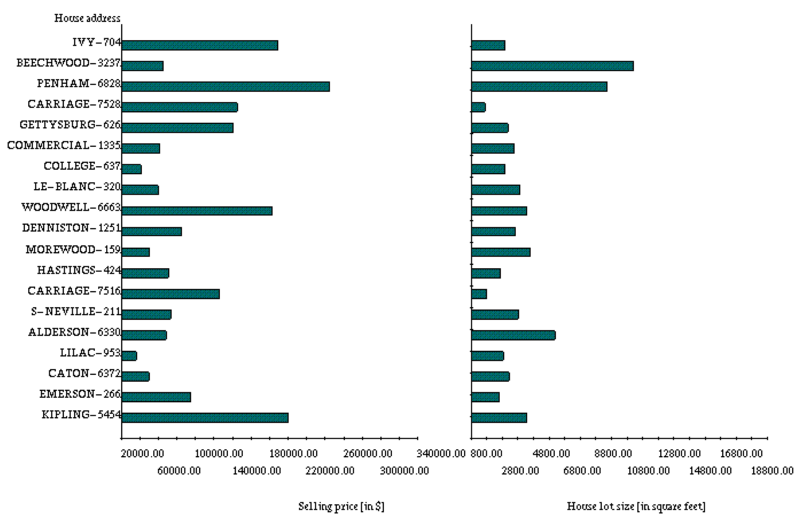


Figure III-6: *House* data-type to *bar* graphical-class mapping applied to the entire visualization. Both chart regions within the visualization inherit this mapping relationship. The *x-axis* of left chart shows *selling-price*; *x-axis* of right chart shows *house-lot-size* and *y-axis* of both charts show *house-address*.

For example, applying an *object mapping* function that maps *house-data-type* to the *bar-graphical-class* onto a *visualization* container will cause all the *region* containers within the *visualization* container to inherit that mapping (e.g. Figure III-6). I.e. all *house* data concepts associated with each *region* container in Figure III-6 gets mapped to *bar* graphical objects based on the mapping transform attached to their parent *visualization* container. Alternatively, we can apply separate object mappings to each of the regions in Figure III-6 instead of just applying one object mapping to the entire visualization container. For example in Figure III-7 we have applied a *house-data-type*::*bar-graphical-class* object mapping to the left region and a *house-data-type*::*mark-graphical-class* object mapping to the right region.

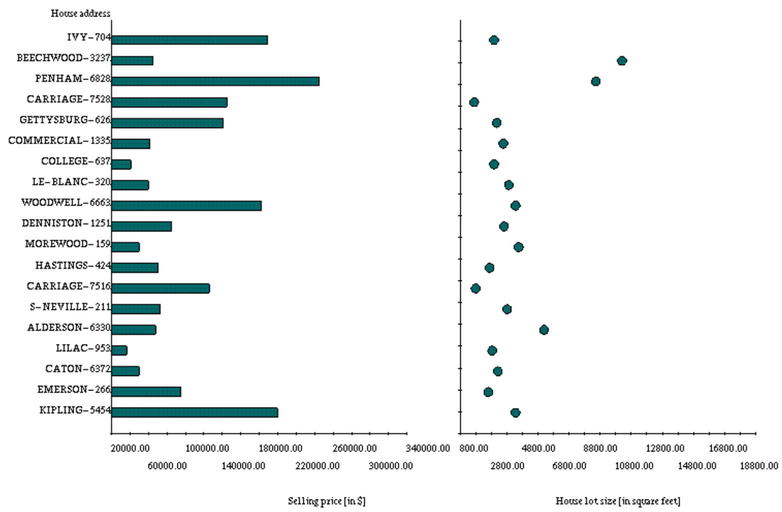


Figure III-7: The same visualization design as Figure III-6 except that a *house* data-type to *bar* graphical-class mapping is applied to the left region and a *house* data type to *mark* graphical-class mapping is applied to the right region.

Mappings very commonly get attached to container objects (e.g. *visualization container*, *space container* or *region container*) as was just discussed. However, limiting object mapping transforms solely to container objects, only allows us to control how data concepts are mapped to graphical objects on a region by region basis (i.e. we cannot associate subsets of data concepts within a given region to different mapping transforms). In the Table Lens [*Rao*, *1994*] technique, the data concepts that appear within the table lens are mapped to the *text* graphical class, and all other data concepts are mapped to the *bar* graphical class. The objects within the *lens* may span multiple column *regions* and only include a subset of objects within each region. This Table Lens operation therefore cannot be achieved with a *region* scope mapping transform To enable Table Lens type mapping, we allow the mapping functions in our framework to be applied to container objects as well as to particular graphical objects within those containers. Note that in order for there to be graphical objects in the first place, we must begin by applying an object mapping transform to a container object. We can then refine the appearance of particular graphical objects within the container by remapping them to a new graphical class. For example in Figure III-8 we have remapped some of the graphical objects in the left-*region* container of Figure III-6 to the *mark* graphical class.

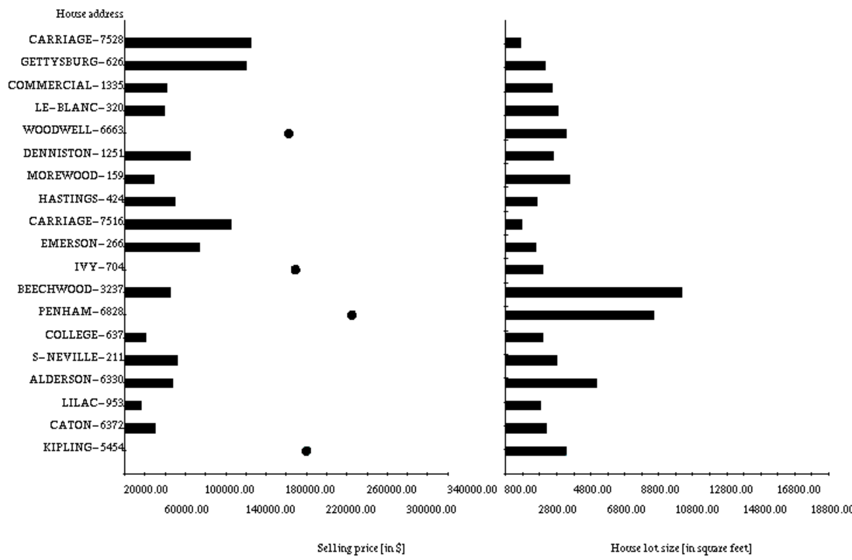


Figure III-8: The same visualization design as Figure III-6 except that a *house* data-type to *mark* graphical-class mapping transform is applied to particular graphical objects in the left chart including *Woodwell-6663, Ivy-704, Penham-6828,* and *Kipling-5454*.

The second primary mapping function is the *attribute mapping* function. Attribute mappings are used to express data attribute values through the use of graphical properties. All visualizations contain a set of data attribute (e.g. *net-profit*, *number-of-rooms*) to graphical property mappings (e.g. *position*, *color*) for illustrating the trends and relationships of data values visually[4]. As is shown in Table III-2, attribute mappings have four input arguments. The first two arguments are the data attribute and graphical property we want to link. The third argument specifies the scope of the attribute mapping. The scope of an *attribute mapping* is specified in the same way as the scope of an *object mapping*. I.e. *attribute mappings* can be applied to container objects or to graphical objects. Finally the last argument (the *mapping-effect* argument), indicates whether the mapping function allows subsequent changes in data values to affect their related graphical values and vice versa. There are four types of mapping-effect operators: *forward*, *backward, both* or *none*. *Forward* allows subsequent changes in data values to be reflected in their corresponding graphical values. *Backward* allows subsequent changes in graphical values to be reflected in

---

[4] *Attribute mappings* usually map data attribute values to graphical property values linearly. However, in some cases we may need to use a non-linear function or a modified linear function to take into account peculiarities of the human visual system. For example Teghtsoonian [*Teghtsoonian*, 1965] found that the perceived *area* is typically the actual area raised to a power of .8. Similar discrepancies arise in *length* and *diameter* judgements. We currently do not deal with capturing these different attribute mapping functions in our framework but such extensions would not be overly difficult to implement.

their corresponding data values. *Both* refer to a combination of *forward* and *backward* effects and finally *none* does not allow updated data or graphical values to propagate either way. For example, we may use the *backward* mapping-effect to link the size of a mark to the *selling price* data attribute. This allows us to change the underlying data (i.e. the *selling price* attribute values) by controlling their corresponding graphical representations (i.e. by changing the size of the *mark* graphical objects).

In addition to *object* and *attribute mapping* functions, the mapping transform class also includes the *add-object* and *delete-object* functions. These functions allow us to change the scope of preexisting object and attribute mappings by changing the data concepts that a container object is associated with. The *add-object* transform **associates** a given set of data concepts with a container object and the *delete-object* transform **disassociates** a given set of data concepts from a container object. Associating new data concepts to a container object will cause those data concepts to be added to each sub-container within the container object. The added concepts will then be mapped according to the mapping transforms associated with the lowest container class. For example attaching a new set of houses to the visualization container in Figure III-7 would cause those new data concepts to be mapped to *bars* in the left region and *marks* in the right region. Furthermore, the new data concepts will have their *house-selling-price* data attribute mapped to the *length* graphical property in the left region, their *date-sold* data attribute mapped to the *mark-x-position* in the right region, and their *house-address* data attribute mapped to the *y-axis* in both regions.

III-1.2.2.2 Data and Graphical Transforms

Unlike mapping transforms, which change elements of one class (i.e. data), to another (i.e. graphical), data and graphical transforms change elements of a single class to different forms within that class. There are five classes of data and graphical transforms in our framework (shown in Table III-3): *unary-functions, binary-functions, summary-functions, assign-function* and *specialized-functions*. These data and graphical transform functions are applied to change existing data and graphical values or to generate new values. There are three primary ways in which these transforms are applied:

1. To change or summarize the values of a single data attribute or graphical property.

$$G_{att1} \rightarrow G_{att2}$$

$$D_{att1} \rightarrow D_{att2}$$

2. To convert one type of attribute or property to another.

$$G_{att1} \rightarrow G_{att2}, \quad att1 \neq att2$$

$$D_{att1} \rightarrow D_{att2}, \quad att1 \neq att2$$

3. To derive a new attribute or property based on multiple existing attributes and properties.

$$G_{att1}, G_{att2}, \ldots, G_{att\_n} \rightarrow G_{att\_m}$$

$$D_{att1}, D_{att2}, \ldots, D_{att\_n} \rightarrow D_{att\_m}$$

for any n and m, where $n > 1$, and att_m is a new attribute

1. *Change the values of a single data attribute or graphical property*

Example techniques that fall within this first class include most of the operations in the SDM [*Chuah, 1995b*] system that are used to improve the readability of a visualization.

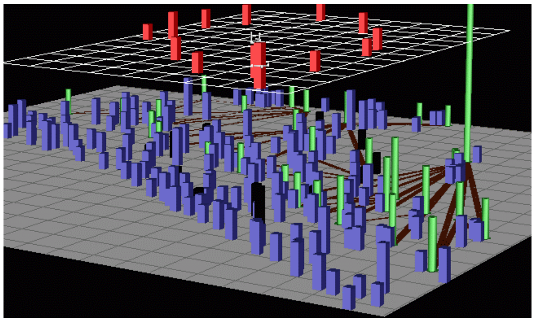


Figure III-9: SDM lift objects technique

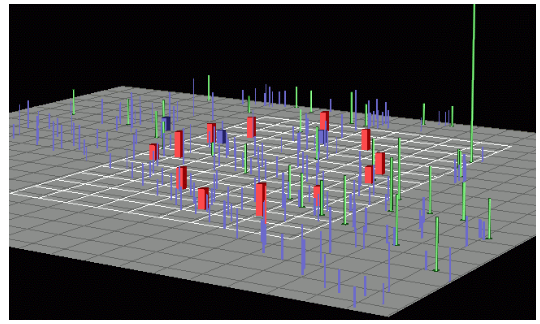


Figure III-10: SDM thin objects technique

Some examples include the *lift objects* technique that allows users to lift selected objects over and above the other objects in the graphical scene so that they are visually more salient and less occluded (shown in Figure III-9). This *lift objects* technique is a $G_{z\_position} \rightarrow G_{z\_position}$ graphical transform; it changes a set of *z-position* graphical values by adding a constant to it. Another example SDM technique that falls into this class is the *thinning objects* technique that allows users to change the width of contextual graphical objects so that they do not occlude the focus objects (shown in Figure III-10). This example is a $G_{width} \rightarrow G_{width}$ graphical transform. Other examples include the feedback operations in the dynamic query [*Ahlberg, 1992*] and painting techniques [*Becker, 1987*], which is a $G_{color} \rightarrow G_{color}$ transform. Such feedback operations commonly use the *assign* graphical transform to set color values of focus objects to a salient constant value. All the transform functions described above change the values of a single graphical property by adding or subtracting constants to/from the value set or by setting the value set to a constant.

2. *Convert one type of attribute or property to another*

A slightly more complex use of graphical and data transforms is to apply the values of one attribute or property to determine the values of another. A common application of this class of functions is to do value conversions. For example the data attribute *temperature in Kelvins* can be used to calculate the data attribute *temperature in Fahrenheit*, e.g. $D_{Kelvins} \rightarrow D_{Fahrenheit}$.

Another application of this class of transforms is to link properties so that they will change in tandem. For example we may want our rectangle-shaped marks to always appear as squares. In order to achieve this we may map $G_{x\_range} \rightarrow G_{y\_range}$ using the *assign* function. Once we have done this, any manipulation function that causes the *x-range* property to change will cause a similar change in the *y-range* property.

This class of transforms is also an integral part of expressing animation. Animation is achieved by linking physical time (G_{time}) to a graphical attribute. For example, in order to make the size of marks in a display increase with time, we define a $G_{time} \rightarrow G_{size}$ transform.

Other example techniques that fall within this category include summarization techniques. These techniques are used to summarize or aggregate data or graphical elements so that the visualization is simplified and users can interpret the relationships shown by the graphic design more quickly and effectively. For example, instead of representing all the *house-selling-price* values in our house database, we can group the house concepts by *neighborhood* and show the average/mean *house-prices* by neighborhood ($D_{house\_price} \rightarrow D_{mean\_house\_price}$). In the *PAD++* system, file objects are summarized or aggregated together (i.e. summarized) when there is very little space allocated to them ($D_{file} \rightarrow D_{aggregate\_file}$). When users *zoom* in on an aggregated file object, it gets split up into its individual components.

The final operations that fall within this category are specialized functions that are used to extract meta-data from a set of values. Some examples include determining the order of the values (e.g. *sort* transform: $D_{attribute} \rightarrow D_{sort}$) or the number of times a particular value appears (e.g. *count*: $D_{attribute} \rightarrow D_{count}$). This meta-data allows users to analyze additional structural information about the existing data set.

3. <u>Derive a new attribute or property based on multiple existing attributes and properties</u>

Data and graphical transforms may also be used to calculate derived values from multiple attributes and properties. For example we can determine the *gross profit* of various *company* data concepts *by subtracting* their *total-cost* attribute values from their *total-sales* attribute values. In this case a new data attribute, *gross-profit* is generated based on existing data attributes *total-cost* and *total-sales* ($D_{total\_sales}, D_{total\_cost} \rightarrow D_{gross\_profit}$). Another example is the *SDM-distance* technique where the *x-position* and *y-position* of objects are used to calculate their distance to the user ($G_{x\_position}, G_{y\_position} \rightarrow G_{distance}$). Note that this class of functions is similar to the previous one except that we have multiple attributes resulting in only one attribute (i.e. a *many-to-one* mapping in contrast to the *one-to-one* mapping of the previous section). Thus similar to the previous category we can convert, link, and summarize attributes and properties, as well as compute meta-data.

In Table III-3 we list the data and graphical transform instances in our framework as well as their input and output arguments.

Function class	Function name	Input	Output
Unary Function		1. Unary operator, (*complement, absolute*) 2. Single value set.	Single value set
Binary Function		1. Binary operator, (*add, subtract, multiply, divide*) 2. *n* value sets.	Single value set
Summary Functions	Group objects	1. Single object set, 2. Group object data type.	Group object
	Summarize values	1. Summary operator, (*sum, mean, median, std-deviation, min, max*) 2. Single value set.	Single value
Assign Functions	Assign	1. Single value set containing the destination values, 2. Single value set containing the source values.	
Specialized Functions	Sort	1. Sort operator, (*increasing, decreasing*) 2. Single value set.	Single value set
	Count	1. Single value set containing the values we want to count, 2. Single value set that is being counted.	Single value set

Table III-3: List of data and graphical transformation functions

1. Unary functions

Unary functions take a value set as input and produce a transformed value set. We consider two types of unary operators: *complement* and *absolute*.

The *complement* operator may be applied to:

1) numbers (i.e. quantitative and discrete data attributes)
   (e.g. –10 → 10, or 5.23 → -5.23).
2) boolean values
   (e.g. T → F, F → T)
3) strings
   (e.g. abc → cba, dracula → alucard)

The *absolute* operator may be applied to:

1) numbers
   (e.g. –10 → 10, or 5.23 → 5.23).

2. Binary functions

Binary functions, unlike unary functions, take n sets of values (where $n > 1$) and a binary operator as input. The binary operator is applied to each value set in turn, e.g. $n_1 + n_2 + \ldots + n_m$, where n represents a particular value set and + is the input binary operator. There are four binary operators in our framework: *add, subtract, multiply, divide*. The effects of each operator are shown below.

The *add* operator may be applied to:
numbers
$(x, y) \rightarrow x + y$
strings
(*"abc"*, *"bc"*) → *"abcbc"*

The *multiply* and *divide* operators may be applied to:
numbers
$(x, y) \rightarrow x * y$ or $(x, y) \rightarrow x / y$

The *subtract* operator may be applied to:
numbers
$(x, y) \rightarrow x - y$
strings
(*"abc"*, *"bc"*) → *"a"*

3. Summary functions

In addition to binary and unary compute operators, there is also a set of summary operators (third row of Table III-3). Very often, especially in large data sets, we may want to summarize a set of data concepts and represent the set as a single summary object in order to reduce clutter. This is done by the *group objects* transform which creates a group data element from a set of data concepts. Like the *group objects* operator, the *summary compute* operator is also used for summarization. However, it summarizes a set of values instead of a set of objects. For example, the summary compute operator was used in the *modified value painting* technique (chapter II-2.3) to calculate the mean *house-price* value which is then used in a subsequent object definition threshold function.

4. Assignment function

Unary, binary, and summary functions produce new data and graphical values. To update existing data attribute or graphical property values with new values we use the *assign* function. The *assign* operator takes two value sets as input and assigns the second value set to the first value set. For example suppose we want to update the *house selling-price* data attribute in our database by adding in a new house sales tax value. We can do this by first computing the new *house-selling-prices* using the *addition* and *multiplication* binary operators and then assigning the new values to the old house price attribute values with the *assign* operator.

5. Specialized functions

Finally there are a set of specialized data and graphical transform functions. These functions correspond to common statistical computation operators: *sort* and *count*. The *sort* transform is used to produce a set of ranks based on the numerical, alphabetical or semantic ordering of the input value set. The *count* transform determines the number of times a particular element occurs in an input set. We can either count all the existing elements in the input set or only specific chosen elements.

It is important to note that even though data and graphical transforms use the same functions to transform objects and values, their end goals are very different. Data transforms are used to prepare data concepts and values in a way that is appropriate for our task(s). As described in the previous chapter, data transforms are used for three primary purposes: 1) to calculate derived results, 2) to summarize existing data, and 3) to compute meta-data based on existing information. Graphical transforms, however, are used to provide feedback and to improve the readability of a visualization whose contents are already defined by the data and mapping transform stages.

III-1.2.2.3 Rendering Transforms

Rendering transforms are used to map a graphical scene onto an output media such as the CRT screen. Currently the only primitive rendering function in our system allows us to access the camera in the graphical scene and change the camera parameters such as *position*, *rotation*, *focal length*, etc. This allows us to navigate (pan, zoom, rotate) within any visualization that is generated. We have left out detailed descriptions of rendering functions because this thesis is only focussed on the use of data processing and mapping functions. For information on distortion rendering techniques refer to Leung et al.'s taxonomy [*Leung, 1994*].

III-1.3 Input & Output Translation Functions

Apart from picking specific object definition and transformation functions, we must ensure that the arguments of a source function match the arguments of a destination function as was described in step 2 of the instantiation augmentation process[5] outlined earlier in this chapter. To achieve this, there are translation functions that allow data and graphical elements to be queried for related attributes, properties, and relationships. For example we may query a set of data concepts for the set of graphical objects that are used to represent them, or we may query a visualization for the set of data concepts associated with it. Table III-4 shows these translation functions.

[5] The instantiation augmentation process is the five step process for converting functional specifications into instantiable visualization techniques.

Translation function class	Function name	Input	Output
Data & Graphical object translation functions	Get related graphics	Single set of data concepts	Single set of graphical objects
	Get data type	Single set of data concepts	Single set of data types[6]
	Get data concepts	A data type, or a container object	Single set of data concepts
	Get related data	Single set of graphical objects	Single set of data concepts
	Get graphical class	Single set of graphical objects	Single set of graphical classes[7]
	Get graphical objects	A graphical class, or a container object	Single set of graphical objects
Container object translation functions	Get parent	A graphical object or a container object	A container object
	Get children	A container object	A set of container objects
	Get mapped data attributes	A container object	A set of data attributes that are mapped to graphics within the container
	Get mapped data types	A container object	A set of data types that are mapped to graphics within the container
	Get mapped graphical properties	A container object	A set of graphical properties that are mapped to data within the container
	Get mapped graphical classes	A container object	A set of graphical classes that are mapped to data within the container
Object attribute translation functions	Get object attributes	A visualization object (e.g. a data concept, a graphical object, a data type, a graphical class, a container object, or an annotation object)	A set of attributes
Attribute value translation functions	Get values	1. A set of visualization objects 2. An object attribute	A set of values
Value translation functions	Get objects	A set of values	A set of visualization objects
	Get boolean objects	1. A set of visualization objects 2. A set of boolean values	A set of visualization objects
	Get named object	1. A string	A visualization object
	Get type	A visualization object	A string
System wide translation functions	Get all objects	Object type (e.g. data concept, graphical object, visualization, region, etc)	A set of objects

Table III-4: Input and output translation functions

Note that all translation relationships can be queried both ways. For example being able to query for all the graphical objects associated with a set of data concepts (*get-related-graphics)* means that there is a related translation function that allows us to query for all the data concepts associated with a set of

[6] Data types capture characteristics of similar data concepts that contain the same data attributes and relationships.

[7] Graphical classes capture characteristics of similar graphical objects that have a common visual appearance and contain the same graphical properties. Some common graphical classes include *bar-class, mark-class, node-class*, and *interval-bar-class*.

graphical objects (*get-related-data*). Or being able to query a data concept for its related data type (*get-data-type*) means that there is a related translation function for querying a data type to get all the data concepts associated with it (*get-data-concepts*). This symmetry allows us to easily move back and forth between the data and graphical realms as well as between different object types within each realm so that we can flexibly build a wide range of visualization techniques using transformation functions that are in any order.

Object translation functions allow us to query a set of data concepts for their related graphical objects and vice versa. We may also query data concepts for their data type or for the attributes they contain (i.e. *get-attribute-values* function). Similarly we may access the graphical class and properties of graphical objects.

Container functions allow us to query a container for other container objects based on the hierarchical relationships described in section III-1.1.2 (through the *get-parent* and *get-child* functions). In addition, the *get-graphical-objects* and the *get-data-concepts* functions (listed in the *data & graphical object translation function class*) allow us to access all the data or graphical objects associated with a container. We may also query container objects for all the data attributes, data-types, graphical properties and graphical classes that are currently involved in an object or attribute mapping (e.g. *get-mapped-attributes, get-mapped-data-types, get-mapped-properties* and *get-mapped-graphical-classes*).

Object attribute and attribute value translation functions allow us to query data concepts, graphical objects, and container objects (e.g. *visualization, graphical space, region*) for the set of attributes and values associated with them. For example a *house* data concept may contain the *house-price* and *date-sold* attributes, a *mark* graphical object may contain the *x-position* and *color* attributes, and a *visualization* container may have the *size, x-position* and *y-position* attributes. These attributes and values can be extracted from their corresponding objects by using the *get-object-attributes* and *get-values* translation functions.

Value translation functions allow operations on value arguments. For example, values can be queried using *get-objects* value translation functions to obtain the objects containing those values. The *get-boolean-objects* function filters an input object set based on a set of **boolean values**. Specifically, only those objects that have a corresponding *True* value in the boolean value set are included in the function output. The *get-named-object* function returns the visualization object that corresponds to the input **value string**. The *get-type* function returns the class to which an object belongs (e.g. *data concept, graphical object, visualization object,* etc).

Finally there is a set of global translation functions that allow us to query for system wide state such as getting all the data concepts or graphical objects within the entire visualization system (e.g. *get-all-data-concepts, get-all-graphical-objects*). Similarly we may also query for all existing data types, graphical classes, regions, spaces, and visualization containers.

III-1.4 Input-devices

To characterize input-devices, our framework uses Foley et al.'s [*Foley, 1990*] description of input-devices, which consists of three levels of design: *lexical design*, *syntactic design* and *semantic design*. *Lexical design* refers to how input primitives are derived from basic hardware functions. Input primitives include all physical device signals such as mouse clicks, key presses, etc. *Syntactic design* consists of a set of rules by which primitive input units can be composed or joined to form ordered sequences of inputs. For example a series of mouse movements, mouse clicks, and mouse releases are required for specifying the syntactic design for a *bounding-box*. Devices that are built from a well-defined sequence of physical device signals are also referred to as *virtual devices*. Note that while syntactic constructs describe how a device may be manipulated, they do not define its meaning (i.e. its semantics). *Semantic design* defines the meaning of a syntactic construct. For example, *mouse clicks, bounding-boxes* and *sliders* can all be used to define a *selection* of objects. In this case the semantics of the device is the selection of objects, while the actions used to achieve this selection could take multiple syntactic forms (i.e. *mouse clicks, bounding-boxes* or *sliders*). Similarly, a syntactic form can have several meanings. For example a *bounding-box* can be used for selecting a set of objects or for defining a set of values, one at each of its vertices. The list of input-devices considered in this thesis are listed in Table III-5.

Input-device	**Input-device trigger event (syntactic)**	**Required initialization attributes (semantic)**	**Output arguments (semantic)**
Bounding-box	Mouse up	Mouse button that activates the device (either *left, middle,* or *right*)	Annotation objects, Graphical objects, Region, Vertex values
Mouse click	Mouse up	Mouse button	Annotation objects, Graphical object, Region
Option menu, Scroll list, Radio buttons	Double click	A set of strings to put into the device	A single or set of strings
Text box	Enter key	Label	A string
Button	Mouse up	Label	A boolean value
Slider, Dial	Mouse move	Minimum and maximum range of device	One or more values

Table III-5: Input-device Query functions

To define an input-device in our framework, we must define it in terms of the three levels of design described above (physical, syntactic, and semantics). First of all, we define the types of input signals available. In this thesis we only consider input from two physical devices, the *mouse* and the *keyboard*. At the syntactic level, we define the input-device's appearance (i.e. menu, slider, etc) and the input primitives used to control it. This includes the *trigger event*, i.e. the physical event that triggers an update of the input-device. For example *double clicking* would update a *menu*, or a *mouse release* event would update a *bounding-box*. Finally at the semantic level, we define all the initialization arguments needed, and all the output arguments the device is capable of producing. For example, in a slider device class there are two important initialization attributes: the min and max values of the slider. The slider can then be queried for the value(s) marked within it.

III-1.5 Summary

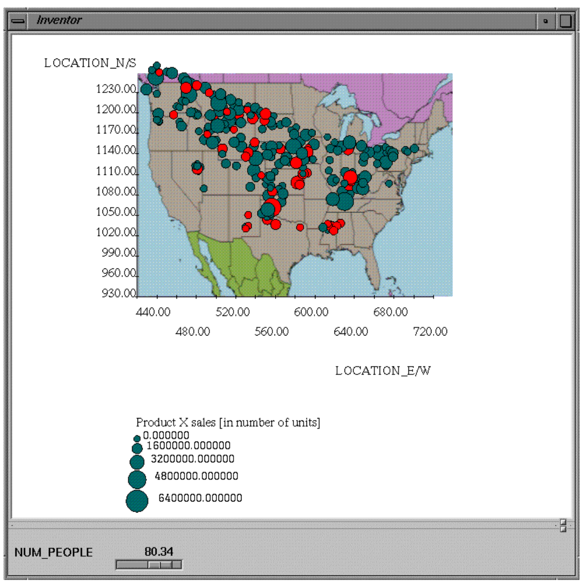
In this section we presented detailed lists of all the visualization objects, functions, and input-devices our automatic system may operate on. We mainly focussed on characterizing data and mapping transforms because the focus of this thesis is on integrating data processing and mapping functions. We also described some graphical transforms because the primitive computations they use are identical to data processing operations (section III-1.2.2.2). We left most of the rendering functions to be specified in future work. It is important to reiterate that the list of argument types, functions, and input-devices provided in this section is not complete. The functions included were chosen because we felt that they were effective for capturing the functions of current information visualization techniques. Now that we have presented all the primitive operators within our framework, we can also them to systematically explore the instantiation space of interactive techniques (sections III-2 and III-3). These sections may be skipped if the reader is only interested in the automatic design aspects of this work.

III-2 Visualization Techniques Instantiation Space

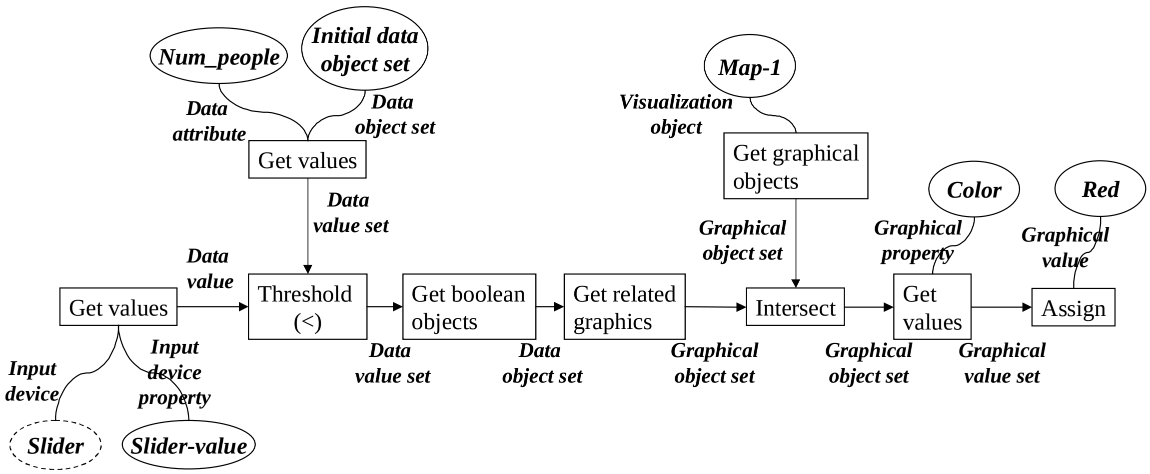
At the end of the previous chapter we show how our framework can help us explore new techniques within the functional design space and improve on existing techniques by combining their functionality. In this section we explore the instantiation design space. Unlike exploration in the functional space which may change the semantics of a technique, exploration in the instantiation space only changes the structural aspects of the technique such as how the technique gets manipulated, which aspects of the technique users get to control, what type of feedback is used, etc. Changes to the instantiation description **do not** change the semantics of a technique. In this section we present an example visualization technique and show how making changes to its instantiation specification can improve its effectiveness. We explore the instantiation space in greater detail in appendix B-4 where we systematically lay out all the alternative instantiation designs for the dynamic query slider technique. In appendix B-5 we explore the instantiation design space for a set of current visualization techniques.

In this example we analyze data from a set of distributors for a hypothetical company in the *United States*. Each mark on the maps in Figure III-11, Figure III-12, and Figure III-13, represents a distributor. The *location* of a mark encodes the geographic location of a distributor and the *size* of a mark encodes the total number of *product-X* units sold by that distributor.

At the bottom of the interface in Figure III-11a is a *dynamic query slider* technique that allows users to highlight distributors based on number of employees (*num_people*). This is done by marking a *num_people* threshold value on the *slider*. Subsequently, all distributors whose *num_people* exceed the marked threshold will get highlighted *red*. The slider visualization technique described above can be defined by the instantiation specification in Figure III-11b. In this specification, we query the slider input-device for the threshold value marked within it. This value is piped into a *threshold* function that filters all of the *num_people* data values. The *threshold* function returns a set of boolean values that we convert back to data concepts using the *get-boolean-objects* translation function. At this point we have a set of distributor concepts whose number of employees are below the threshold value. We query these distributor data concepts for all the graphical objects used to represent them. These graphical objects are then intersected with the graphical objects in the map visualization. The intersection operation is necessary to localize the highlight effect to only the map visualization. Finally the intersected set of graphical objects are colored *red* using the *assign* graphical transform operator.



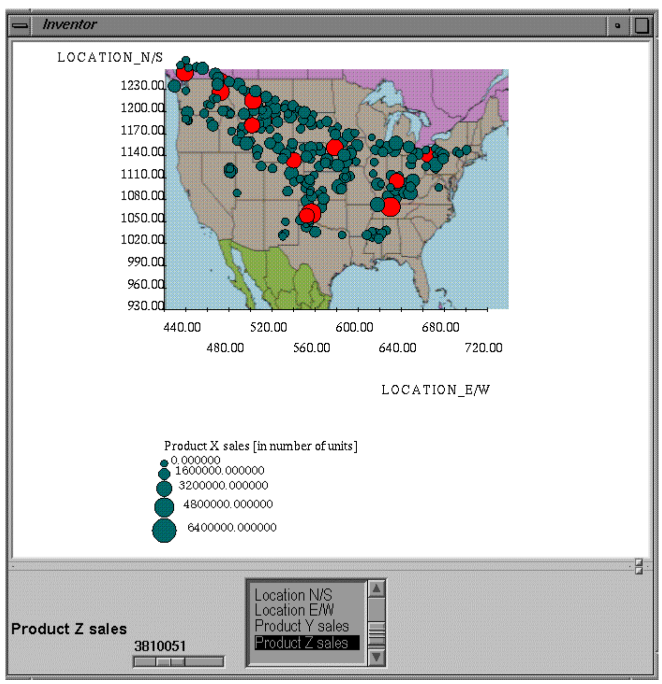
(a) Each mark in the map encodes a *distributor* for a hypothetical company. The *x-axis* and *y-axis* encodes the geographic location of the *distributor*. *Size* encodes the total number of units sold for *product-X*. The slider at the bottom of the interface allows users to select *distributors* based on the number of employees (*num_people*) there. Selected *distributors* are highlighted in *red*.



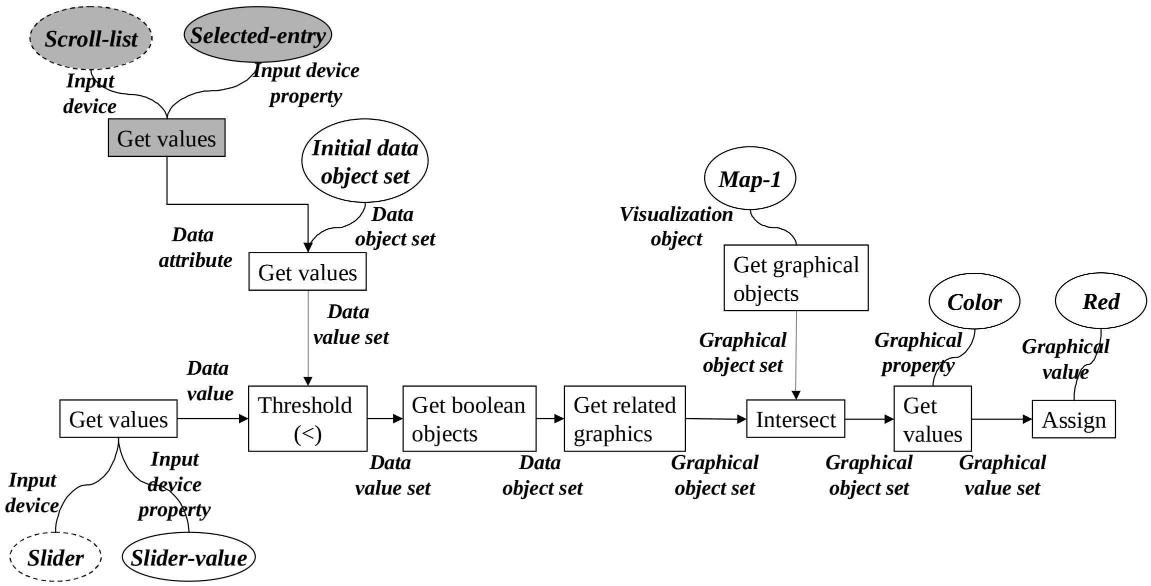
(b) Instantiation specification for the dynamic query slider technique in Figure III-11a

Figure III-11: Example of a dynamic query slider technique that allows users to select various *distributor* data concepts based on the number of employees (*num_people*) working at each site.

A problem with the visualization technique in Figure III-11 is that users can only highlight distributors based on their *num_people* attribute. In order to relax this constraint, we add a new input-device as shown in Figure III-12.



(a) Similar interface to Figure III-11a except that here users get to control a *scroll-menu* in addition to the *slider* in order to pick the current constraint data attribute. Currently, the *product_Z_sales* attribute has been picked on the *scroll-menu* and therefore it appears as the constraint attribute next to the *slider*. Therefore the highlighted objects are those *distributors* with *product_Z_sales* less then 3810051 units.



(b) Instantiation specification for the visualization interface in Figure III-12**a. This specification is similar to the one in** Figure III-11**b except that a *scroll-menu* has been added to allow users to pick the constraint attribute. Changes made to the specification in** Figure III-11**b are shown in gray here.**

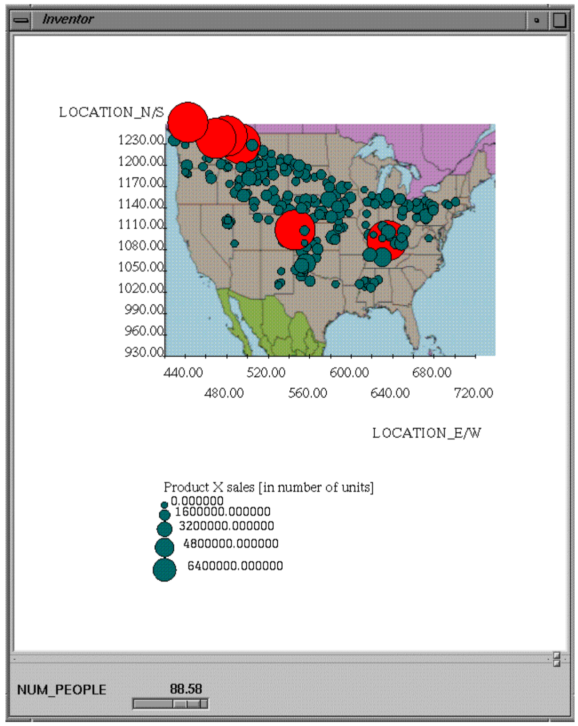
Figure III-12: Example dynamic query slider technique with selectable data attribute constraint

In Figure III-12b we have added a *scroll-menu* input-device which allows users to supply the data attribute on which to perform the threshold operation. Alterations made to the specification in Figure III-11b are shown in light gray in Figure III-12b. The resulting interface (Figure III-12a) is identical to Figure III-11a except that there is a *scroll-menu* below the *slider*, which contains all the distributor attributes (e.g. *location-n/s, location-e/w, product-X-sales, product-Y-sales, product-Z-sales*, and *num_people*). Through this interface users can not only pick the threshold value but also the threshold attribute.

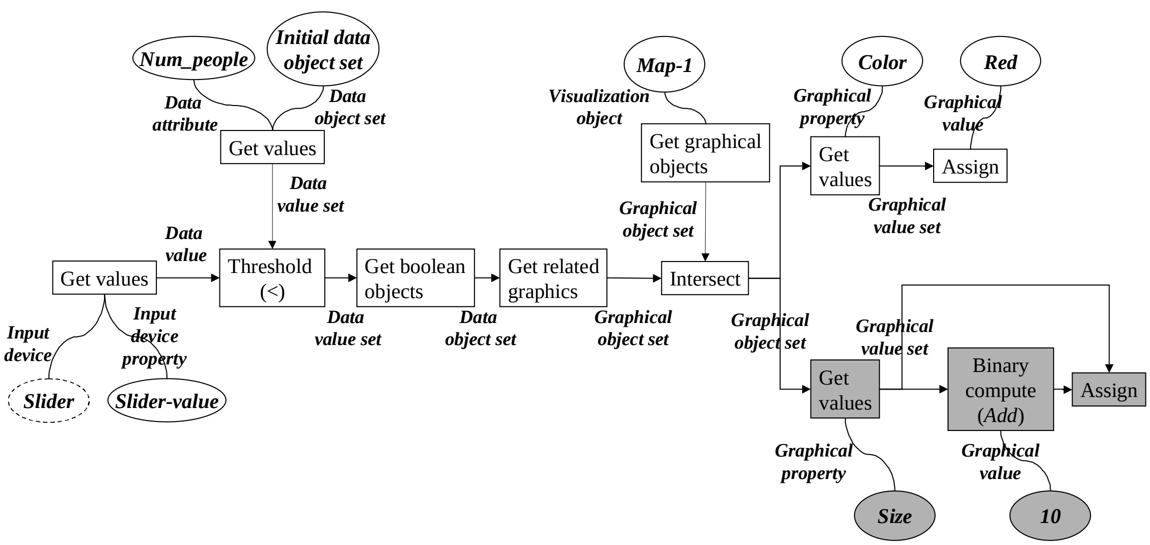
Another problem with the slider highlight technique in Figure III-11 is that occasionally, object *coloring* alone does not provide sufficient feedback because the colored objects may be occluded, thereby making them difficult to find in the visual display despite their coloring. One way to solve this problem is to both enlarge the objects as well as color them. In this way, the highlighted objects become much more salient. We do this simply by adding some new graphical transform operators to the technique specification in Figure III-11b (changes are shown in light gray in Figure III-13b). This new specification will cause the selected objects to be colored *red* as well as enlarged as is shown in Figure III-13a[8].

---

[8] Note that enlarging the selected objects is sometimes undesirable because they tend to occlude each other or the objects around them. We can alleviate this problem somewhat by changing the draw order of the objects so that the smaller elements are drawn last.



(a) Similar interface to Figure III-11a except that here the selected objects are both colored *red* as well as enlarged in order to increase saliency.



(b) Instantiation specification for the visualization interface in Figure III-13**a. This specification is similar to the one in** Figure III-11**b except that some additional graphical transform functions are included to increase the size of the selected objects. Changes made to the specification in** Figure III-11**b are shown in gray here.**

Figure III-13: Example dynamic query slider technique with color and size feedback on the selected objects

The examples in this section show that refining a visualization technique at the instantiation level allows us to improve the effectiveness with which a technique can be applied to the current task situation and to our current preferences. This section also illustrates some of the design decisions that must be made by our automatic designer and by human designers when creating a visualization technique. We show in appendix B-4 that by following the five steps in the instantiation augmentation process refinements to the instantiation specification can be carried out systematically and effectively. In appendix B-5 we begin analyzing the instantiation space for a set of current visualization techniques. These examples further illustrate the differences between the functional and instantiation levels from a design standpoint and show the applicability of our two level design methodology. Specifically, exploration in the functional space is useful when we want to create new techniques that have unique or additional functions. On the other hand exploration of the instantiation space is useful when we want to refine a technique to better suit the current hardware, user preferences or task conventions, without altering its underlying functionality. Note that the customization examples shown in this section (Figure III-11, Figure III-12, Figure III-13) do not alter the general functionality of the slider technique. The function of the slider technique, which is to allow users to select a set of data concepts, based on their attributes and then to highlight their corresponding graphical objects, remains the same in all three specifications.

III-3 Evaluation of Framework

We evaluate our framework based on three criteria: completeness, coverage, and practicality. *Completeness* refers to whether the framework is capable of expressing all visualization techniques. *Coverage* refers to whether the framework can be applied to a wide range of visualization types (e.g. bar charts, scatter-plots, 3D-displays) and input-devices (e.g. *menus, bounding-box, radio-buttons*). And finally we argue for *practicality* in three ways: 1) the framework reduces the cost of task tailoring; 2) the framework provides a new design methodology; and 3) the framework allows systematic exploration of the visualization techniques design space.

III-3.1 Completeness

Completeness refers to whether the framework is capable of expressing all visualization techniques. At the end of the previous chapter we showed that our framework can express many current visualization techniques. The framework, however, is not complete, and can never be so because as new visualization methods and metaphors are created the framework would need to grow to include these new techniques. It is thus important for the framework to be flexible and easily extensible.

Our framework supports both flexibility and extensibility because it splits the design into two different levels (functional and instantiation). We anticipate that framework extensions will commonly occur only at the instantiation design level because at that level, we are more concerned with input-device specifics and visualization function inputs and outputs. In contrast, the functional level deals with abstract function classes and composition operators, which tend to be less volatile. Changes made to the instantiation design level would generally not affect the functional design level, so framework alterations should be fairly localized. Secondly, the framework is based on a compositional language that allows us to generate a wide space of designs with relatively few primitives. Future extensions should be able to capitalize on this compositional language and leverage off of pre-existing object definition and transformation functions so that only a few primitives need to be added to increase the expressive capability of the framework significantly. Finally, one of the design decisions was to use object definition and transformation functions of lower granularity (i.e. just simple mathematical functions). This makes it easier to reuse and compose these functions, even with new methods and metaphors. This lower granularity level comes at the price of more specification; however we believe that much of the specification can be automated. In addition partial specifications can always be saved and reused so that we only need to declare them once.

III-3.2 Coverage

Our interactive framework can be applied to a wide range of traditional visualizations (e.g. *charts*, *maps*, *tables*) and direct manipulation input-devices and widgets (e.g. *mouse clicks*, *keyboard presses*, *menus*, *radio-buttons*, etc). The examples in this chapter and the previous chapter show the use of a fairly wide range of visualization types including *maps* (Figure III-11), *charts* (Figure III-2), *tables* as well as input-devices including *sliders* (Figure III-11), *drag and drop*, and *menus* (Figure III-12). This is achieved by building our language based on previous work that have characterized a wide range of data [*Mackinlay, 1986a, 1986b; Roth, 1990*], visualization elements [*Mackinlay, 1986a, 1986b; Roth, 1994; Chuah 1995*], and input-devices [*Card, 1990*]. We do not deal with more complex input-devices, such as two-handed input-devices and speech; however such techniques can be relatively easily integrated into the framework by specifying them according to the three input-device levels described in section III-1.4. Framework generality provides flexibility in design, however, it also raises a big concern, namely how to pick the "best" visualization objects or input-devices from the wide range of choices available. In order to make this decision we must carefully consider our data, our media, and our task [*Bertin, 1983; Tufte, 1983; Mackinlay, 1986a, 1986b; Casner, 1991*]. Choosing the "best" visualization design for our task and data is the topic of discussion in the next chapter.

III-3.3 Practicality

Our framework is practical for the following three reasons.

III-3.3.1 Reduces Cost of Task Tailoring

Our framework provides designer with an easy means of combining common transformation functions to form visualization techniques so that they can plug and play with different visualization effects to suit to suit their design goals. The example shown in section III-2 illustrates this by showing how the dynamic query slider technique can be easily expanded in several simple steps to solve some of the limitations found in the original technique.

III-3.3.2 Provides a New Design Methodology

Our design methodology is based on two different levels of abstraction. The functional level of abstraction (outlined in the previous chapter) is concerned with the goals of a technique while the instantiation level of abstraction (outlined in this chapter) is concerned with the form or look of the technique. By dividing the design process into these two levels, designers can concentrate on functionality first without having to worry about specifics like the *color* to use, whether to use a combined input-device, or whether to use *size* instead of *shape*. At the functional level designers can focus on issues such as whether the functions chosen are capable of solving the task(s) well, whether the functions combine well together, and whether sufficient feedback is provided. Also by concentrating on functionality, designers may notice similarities among techniques that they previously considered to be quite different due to superficial differences. This will hopefully encourage more functional reuse among different techniques. This two-tier design methodology allows designers to focus on different aspects of the design process without complications from other unrelated parts. By doing so we are ensuring that their choice of function is more driven by task concerns rather than by media and device restrictions, which should be dealt with separately.

III-3.3.3 Allows Systematic Exploration of the Visualization Techniques Design Space

The visualization techniques design space can be explored based on the two design levels outlined above: functional and instantiation. Exploration at the functional level involves developing new object definition and transformation functions, as well as combining existing functions in new ways to derive new behaviors. We did this in the previous chapter, which showed how techniques with different functionalities can be combined. Exploration at the instantiation level, on the other hand, involves picking which input-devices, graphical properties (*color, shape, position*), data attributes, or graphical elements to use for a technique. In section III-2 we showed some example instantiation specifications and how these specifications may be varied to produce interesting design alternatives. Note that design variations at the instantiation level do not change the functional characteristics of a technique. The functional space for visualization techniques is bound by the object definition, transformation, and composition classes available. The instantiation space for a given technique is bound by the original functional design of the technique and by the five classes of design changes in the instantiation augmentation process.

III-4 Conclusion

In this chapter we describe the instantiation design of visualization techniques. We show how instantiation details can be added onto functional visualization technique specifications so that they may be rendered as an active visualization interface. Just like the functional level, the instantiation level may be explored in a systematic fashion. Through a series of examples in sections III-2 we show some design variations that may be derived from existing visualization techniques.

We also evaluated the entire framework based on three criteria: completeness, coverage and practicality. In terms of completeness the framework is able to express many current techniques but because of its nature can never be fully complete. It is however easily extensible. In terms of coverage the framework allows various techniques to be integrated with a wide range of visualizations and physical and virtual input-devices. Finally, we argued for the practicality of the framework by showing that it a) allowed easy task tailoring b) provided a new design methodology based on two levels (functional and instantiation) and c) allowed the systematic exploration of the visualization techniques design space. In appendix B-6 we discuss more advanced visualization technique issues that occur when we integrate multiple visualization techniques within a common workspace.

Our analysis of the visualization techniques design space show us that there are many visualization technique alternatives for achieving a single data analysis task or problem. Because of the enormous number of design alternatives, it can sometimes be difficult and time-consuming to test out all the design variations. An automatic design system would help designers create and generate their prototypes more quickly and easily. Such a system however requires the framework, which we have laid out in this chapter and the previous one. Our visualization technique framework provides an automatic design system with a language for describing visualization techniques and a systematic methodology for creating and exploring the visualization techniques design space.

This chapter and the previous chapter described a framework of visualization techniques that had data, mapping, graphical, and rendering transforms. We did not, however, explore the effectiveness of these transform functions. Earlier work on automatic visualization design considered effectiveness criteria for mapping transforms based on data and task requirements [*Mackinlay, 1986a, 1986b; Casner, 1991*]. In the next chapter, we consider effectiveness criteria for making combined decisions about data transforms, and mapping transforms. Specifically, we consider when it might be more useful to perform a task or subtask perceptually by mapping it to graphics, and when it might be more advantageous to let the system internally compute the task through data transforms and only visualize the pre-computed results. In appendix F, we consider the role of graphical and rendering transforms in improving the readability of a visualization. Specifically, we consider readability issues such as occlusion, display density, data dwarfing, and information presence, and how these issues affect the usability of a visualization interface.